Metadata Management

Leo Mark and Nick Roussopoulos
University of Maryland

A self-describing database system uses an active and integrated data dictionary to provide metadata to systems and users. The data dictionary system uses the services of the database system to manage metadata.

A database system should support a rich variety of metadata describing and controlling the management and use of data. Few database systems today provide even rudimentary integrated metadata management facilities.

This article presents a self-describing database system. Its active and integrated data dictionary system provides the only source of metadata to users, programs, and the database system, and uses the services offered by the database system for metadata management. We focus on the design of a self-describing metaschema and a formalism for specifying some of the operations controlling the evolution of database schemata.

Architecture for self-describing database systems

A self-describing database system and its environment are illustrated in Figure 1.[1,2] The data mapping control system, or DMCS, supports and enforces two orthogonal dimensions of data description, the point-of-view dimension and the intension-extension dimension.

The point-of-view dimension has three levels of data description: the information meaning described in the conceptual schema, the external data representations described in external schemata, and the internal physical data structure layout described in the internal schema. These three levels of data description result in databases that are flexible and adaptable to changes in the way users view the data and in the way data is stored. This combination of flexibility and adaptability is usually called data-independence.[3]

The intension-extension dimension has four levels of data description:

(1) the information about the data model supported by the database system described in the metaschema,
(2) the information about the management and use of data described in the data dictionary schema,
(3) the information about specific applications described in the application schemata, and
(4) the application data.

Each level of data description in the intension-extension dimension is the intension of the succeeding data description and the extension of the preceding data description. An intension completely describes and controls changes of an extension. All the levels of the intension-extension dimension are described in terms of the same data model. A description of the metaschema is explicitly stored as part of its own extension—it is self-describing. The stored metaschema can be retrieved, but it cannot be changed; it is self-describing, not self-destructing.

The DMCS can be thought of as a DBMS stripped to the bones on the one hand, but still a complete DBMS on the other hand. It supports the set of elementary functions essential to the maintenance of the two dimensions of data description.

0018-9162/86/1200-0026$01.00 © 1986 IEEE

The data language (DL) interface is the data manipulation language for the data model. Because the metaschema is self-describing, all data, including descriptions, is defined, retrieved, and manipulated through the DL interface. No data definition language is needed.

The data management tool box contains software that is plug-compatible with the DMCS through the DL interface. A data management tool supports high-level functions that, although important, are not essential to data management. To produce a plug-compatible data management tool, information about the data model must be retrievable through the DL interface. This is exactly why the metaschema is explicitly stored. Each of these tools will have its own user interface, but each must interface with the DMCS through the DL interface. A database administrator would develop or buy off-the-shelf data management tools, such as schema design aids, software documentation packages, high-level query language processors, report generators, etc., to supplement the elementary functions supported by the DMCS. He or she would develop or buy off-the-shelf definitions of the data dictionary schema and definitions of application schemata for classes of applications.

The internal data language (i-DL) interface is the interface through which all data is passed from the DMCS to the operating system supporting the DMCS.

The two orthogonal dimensions of data description supported by the DMCS, the point-of-view and the intension-extension dimension, are illustrated in Figure 2.

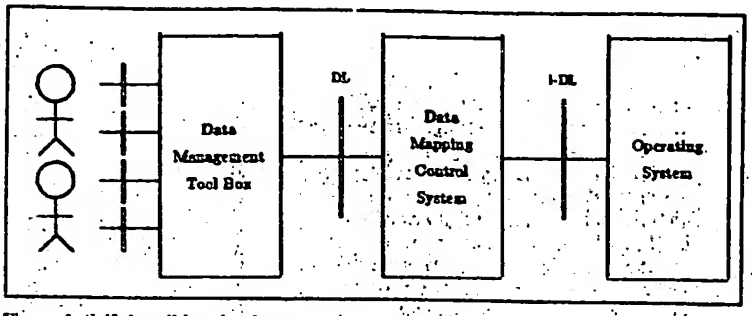

Figure 1. Self-describing database system environment.

Any system in this new architecture is born with data structures to hold the metaschema extension—the data dictionary schema. Initially, the data dictionary schema consists of the stored description of the metaschema. The data dictionary schema can subsequently be expanded into a full data dictionary schema describing and controlling the management and use of application databases. Initially, the extension of the data dictionary schema, the data dictionary data, consists of an empty set of tables.

The architecture supports software plug-compatibility and data plug-compatibility, which gives the database administrator the freedom to choose the tools for data management and define the data management strategy best suited for the enterprise.

The architecture unifies well-established principles and current trends in the areas of database systems and data dictionary systems. It is a generalization of the ANSI/SPARC DBMS Framework, which only considers the point-of-view dimension, and comprises recent ideas on the intension-extension dimension from the International Standards Organization Working Group (ISO/TC97/SC5/WG3) on the conceptual schema and information base. It supplements data model standardization efforts and supplements data dictionary system standardization efforts.

The architecture is based on the notion of self-describing database systems and has matured through several discussions in the ANSI/SPARC Database Architecture

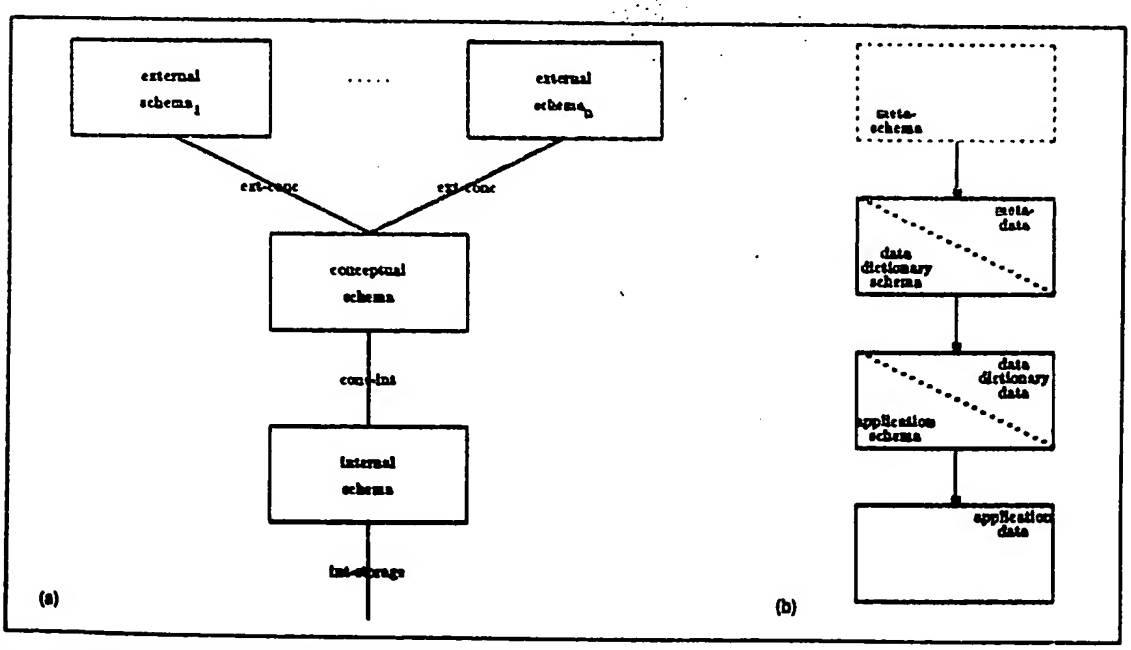

Figure 2. Point-of-view dimension (a) and intension-extension dimension (b).

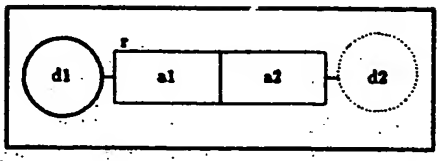

Figure 3. Graphic formalism for data structures.

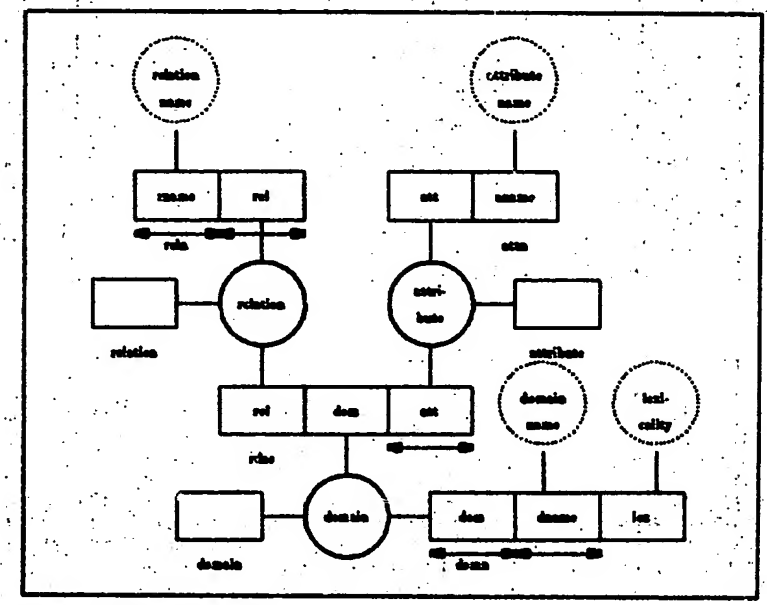

Figure 4. Core metaschema.

Framework Task Group, DAFTG. The architecture has been accepted by ANSI/SPARC and is being considered by ISO as a reference model for database systems in the late 1980's and the 1990's.

The metaschema

We use the relational data model as an example.[4] However, we use a graphic formalism inspired by the object-role data models.[5,6] The symbols in Figure 3 represent the definition of a relation with name r and two attributes with names a1 and a2. The attributes are defined over a nonlexical domain with name d1 and a lexical domain with name d2, respectively.[7,8]

The entities in relational schemata are definitions of relations, domains, and attributes. The entity names used in these definitions are relation names, domain names, and attribute names. Accordingly, we define in Figure 4 the nonlexical domains named relation, domain, and attribute, and the lexical domains named relation-name, domain-name, and attribute-name. The three unary relations named relation, domain, and attribute define sets of entities in existence. The relation named reln defines relationships between existing relations and their relation-name. The relation named attn defines relationships between attributes and attribute-names. The relation named domn defines relationships between domains and their domain-names and lexicality. Finally, the relation named rdas defines the relationships between relations, domains, and attributes. The keys, indicated by double headed arrows, point out attributes, the values of which uniquely identify tuples in the relation. Note that no two relations can have the same name and no relation has more than one name. The same restriction applies to domains and their names. In rdas we indicate that an attribute is a unique entity related to at most one domain and relation.

There are several more constraints involved: the relations relation, domain, and attribute model sets of entities in existence. Therefore, reln[relation] ⊆ relation[relation] and corresponding rules apply for domains and attributes. On the other hand, we insist that all relations have names, so actually reln[relation] = relation[relation]. Also, attribute names must be unique within relations.

These rules, together with the keys and several other rules, will all be specified as part of the operations defined in the metaschema.

Boxes represent metaschema relations. Full circles represent nonlexical domains of surrogates used to model entities. Broken circles represent lexical domains used to model entity-names. Arrows represent keys. The metaschema is so far self-describing. It is defined in terms of relations, domains, and attributes, and its definition can be stored in the database it defines. See Figure 5. (We have omitted the unary relations relation, domain, and attribute.)

Operations in the core metaschema

To make sure that the metaschema completely models and controls all operations on its extension (the data dictionary schema), we define an insert, delete, and modify operation for each relation in the metaschema. From the database administrator's point of view these operations are elementary, but as we shall see in their specification below, each involves several implied operations on update-dependent relations. All the operations we define must be explicitly represented in the metaschema, otherwise the metaschema does not completely model and control all operations on its extension. For the metaschema to be self-describing, it must explicitly model the notion of an operation and control all operations on the specification of operations. We do not consider this expansion of the core metaschema here. For a detailed description, see Mark.[9] Operations are specified in the language we now describe.[10]

Syntax. Each operation is defined by a set of update dependencies, each with the following form:

<op1> —> <cond>,<op2>,...,<opn>.

where <op1> is the operation being defined; <opi>, i = 2,...,n, is either an implied operation or an implied primitive operation; and <cond> is a condition on the database state.

An operation <opi> has one of the following forms:

insert(<relation_name>(<tuple_spec>))
delete(<relation_name>(<tuple_spec>))
modify(<relation_name>(<tuple_spec>)),<relation_name>(<tuple_spec>))

reln

rname: relation name	rel: relation
reln	r1
domn	r2
attn	r3
rdes	r4

domn

dom: domain	dname: domain name	lex: lexicality
d1	relation	non-lexical
d2	attribute	non-lexical
d3	domain	non-lexical
d4	relation name	lexical
d5	attribute name	lexical
d6	domain name	lexical
d7	lexicality	lexical

rdes

rein: relation	dom: domain	att: attribute
r1	d4	a1
r1	d1	a2
r2	d3	a8
r2	d6	a9
r2	d7	a10
r3	d2	a3
r3	d5	a4
r4	d1	a5
r4	d3	a6
r4	d2	a7

attn

att: attribute	aname: attribute name
a1	rname
a2	rel
a3	att
a4	aname
a5	rel
a6	dom
a7	att
a8	dom
a9	dname
a10	lex

Figure 5. Metaschema description stored in metaschema extension.

where the <tuple spec> is a tuple variable for the relation with the name <relation name> and consists of a list of <domain variable>s. The <tuple spec> in <opl> is the formal parameter for <opl>. All the <domain variable>s in the <tuple spec> of <opl> are assumed to be universally quantified. All <domain variable>s in the <tuple spec>s of <opi>, not bound to a universally quantified <domain variable> in <opl>, are assumed to be existentially quantified. All <domain variable>s are in caps; nothing else is.

The implied primitive operations are assert for adding a new tuple in a relation, retract for eliminating one, write and read for retrieving data from the user, new for creating a unique new surrogate, and break for temporarily stopping the system to do some retrieval before giving control back to the system. The implied primitive operations <opi> have the following forms:

assert(<relation name>(<tuple spec>))
retract(<relation name>(<tuple spec>))
write("<any text>"), or write (<domain variable>)
read(<domain variable>)
new(<relation name> (<tuple spec>))
break

The <relation name> used in the operation new must be the name of a unary relation defined over a nonlexical domain. The conditions <cond> are expressions of predicates with the form <relation name>(<tuple spec>). The connectives used in forming the expressions are ∧ (and) and ¬ (negation). In addition, we use the primitive predicates nonvar and var to decide whether or not a <domain variable> has been instantiated.

Conditions can also be used by the user to retrieve data from the system.

Semantics. An operation succeeds if, for at least one of its update dependencies, the condition evaluates to true and all the implied operations succeed. It fails otherwise.

When an operation is invoked, its formal parameters are bound to the actual parameters. The scope of a variable is one update dependency. Existentially quantified variables are bound to values selected by the database system or to values supplied by the user upon request from the database system. Evaluation of conditions, replacement of implied operations, and execution of implied primitive operations are left-to-right and depth-first for each invoked update dependency. For the evaluation of conditions we assume a closed world interpretation.[11]

The nondeterministic choice of a replacement for an implied operation is done by backtracking, selecting in order of appearance the update dependencies with matching left-hand sides. If no match is found, the operation fails.

An implied operation matches the left-hand side of an update dependency if

- the operation names are the same, and
- the relation names are the same, and
- all the domain components match.

Domain components match if they are the same constant or if one or both of them is a variable. If a variable matches a constant, it is instantiated to that value. If two variables match, they share value.

The semantics of the primitive operations are

assert(r(t))
Its effect is r := r ∪ {t}. It always succeeds. All components of t are constants.

retract(r(t))
Its effect is r := r\{t} where all components of t are constants. It always succeeds.

write("text")
It writes the "text" on the user's screen. It always succeeds.

write(X)
It writes the value of X on the user's screen. It always succeeds.

read(X)
It reads the value supplied by the user and binds it to X. It always succeeds (if the user answers).

new(r(D))
It produces a new unique surrogate[7] from the nonlexical domain over which r is defined and binds the value of the variable D to this surrogate. It always succeeds.

break
It suspends the current execution and makes a new copy of the interpreter available to the user, who can use it to retrieve the information needed to answer a question from an operation.

We kept the list of primitive operations minimal to illustrate the concept. It can easily be extended. We should emphasize that the user cannot directly invoke primitive operations.

The execution of assert and retract operations done by the system in an attempt to make an operation succeed will be undone in reverse order during backtracking. This implies that an operation that fails will leave the database unchanged.

The operations. We will only specify the few operations shown in the table in Figure 6 to illustrate the principles. For a detailed discussion, see Mark.[9]

rel/op	insert	delete	modify
domain			
domn			
relation	x	x	
reln	x	x	
attribute	x	x	
attn			
rdes	x	x	

Figure 6. The operations.

```
insert(relation(R))
 → var(R),
   new(relation(R)),
   insert(relation(R)).
 → nonvar(R) ∧ relation(R).
 → nonvar(R) ∧ ¬(relation(R))
   assert(relation(R)),
   insert(rdas(R,_,_)),
   insert(reln(_,R)).
```

Figure 7. Insertion into relation.

```
insert(reln(N,R))
 → var(R),
   new(relation(R)),
   insert(reln(N,R)).
 → var(N),
   write("relation name?"),
   break,
   read(N),
   insert(reln(N,R)).
 → nonvar(N) ∧ nonvar(R) ∧ reln(N,R).
 → nonvar(N) ∧ nonvar(R) ∧ ¬(reln(N,_)) ∧ ¬(reln(_,R)),
   assert(reln(N,R)),
   insert(relation(R)).
```

Figure 8. Insertion into reln.

```
insert(attribute(A))
 → var(A)),
   new(attribute(A)),
   insert(attribute(A)).
 → nonvar(A) ∧ attribute(A).
 → nonvar(A) ∧ ¬(attribute(A)),
   assert(attribute(A)),
   insert(attn(A,_)),
   insert(rdas(_,_,A).
```

Figure 9. Insertion into attribute.

If the variable R in the operation insert(relation(R)) in Figure 7 is uninstantiated when the insertion into relation relation is called, the system produces a new surrogate and proceeds with the insertion. If R is instantiated, two possibilities exist: R is already in relation relation, in which case the insertion succeeds with the database state unchanged; or R is not in relation relation, in which case we insert it. Since all relations must have some attributes and a name, we propagate by triggering insertions into the relations rdas and reln of the tuples (R,_,_) and (_,R), respectively, indicating that the relation surrogate is at this point the only thing we know.

In the operation insert(reln(N,R)), in Figure 8, the first two rules produce or request from the user any uninstantiated variable values. The insertion operation succeeds with the database state unchanged if a relation with that particular name is already in the database. If no relation with the name N exists and the relation represented by surrogate R does not already have a name, we assert the tuple and propagate by inserting R in relation relation.

In the operation insert(attribute(A)) in Figure 9, the first update dependency produces a new attribute surrogate if needed. The second succeeds if the surrogate is already present. The third makes the assertion and propagates through insertions in attn and rdas.

In the operation insert(rdas(R,D,A)) in Figure 10, a new attribute surrogate is produced if needed. If the attribute is not already in rdas and relation and domain surrogates are not provided, they are requested. If attribute name uniqueness within relations is not about to be violated, then the tuple is asserted in rdas. This may cause propagation to relations relation, domain, and finally attribute.

The relation rdas is central to the core metaschema. Since we want domains to exist without being currently used in any relations, the only direct propagation from deleting tuples from rdas is to attribute and sometimes to relation.

The definition of the compound update operation delete(rdas(R,D,A)) in Figure 11 is very special, because it allows the user free use of any combination of unistantiated variables, possibly resulting in multiple tuple deletions. The multiple deletions result from extensive use of recursion, and every single deletion is automatically propagated during the process.

If no tuples matching the actual parameter are present in rdas, the operation succeeds with the database unchanged. If all variables are uninstantiated, we don't allow any change. The operation succeeds with nothing done to the database. The possible combination of instantiated/uninstantiated parameter combinations left are illustrated in Figure 12.

Because the attribute uniquely identifies one tuple, one (domain, relation), all operations with an attribute surrogate cause one tuple deletion from rdas, which is propagated to attribute. If the attribute to be deleted is the last one in a relation, then the deletion also propagates to relation. If only the domain surrogate is given, rdas(_,d,_), then we must delete all uses of that domain in any relation. If only the relation surrogate is given, rdas(r,_,_), then all attributes for that relation are deleted from rdas, implying of course the deletion of the relation too. Finally, if both relation and domain surrogates are given, rdas(r,d,_), then we must delete all uses of the given domain in the given relation.

If the attribute is not present in attribute, the operation delete(attribute(A)) succeeds with the database unchanged. See Figure 13. In the last update dependency, if no value is given, the user is prompted for one and the operation is tried again. In the second update dependency, if a value is given and that value is in attribute, we remove it and propagate by deleting all its names from attn and all its uses from rdas.

It takes only simple arguments to see that if a deletion of an attribute is caused by a previous deletion from rdas, then the propagation to rdas from this update dependency immediately returns with success, because that attribute is not used in rdas anymore (by the very first update dependency).

If, on the other hand, this deletion from attribute is the original one, the propagation back to attribute following the propagation back to rdas will succeed by the first update dependency.

The operation delete(relation(R)) is shown in Figure 14. If a surrogate for the relation is given and the relation exists, it is removed, its name is deleted, and all attribute and domain relationships to the relation are deleted from rdas.

In the operation delete(reln(N,A)) in Figure 15, only the relation surrogate or the name is needed to uniquely identify a tuple to be deleted, so there is no internal recursion in this rule. The propagation of the deletion to relation succeeds with the job done, and the returning call of a deletion from reln stops on the first rule.

In a fully expanded metaschema there will be several more relations and operations modeling and controlling all operations on the specification of operations.[9]

```
insert(rdas(R,D,A))
 ← nonvar(A) ∧ rdas(_,_,A).
 ← var(A),
    new(attribute(A)),
    insert(rdas(R,D,A)).
 ← var(R) ∧ ¬(nonvar(A) ∧ rdas(_,_,A)),
    write("relation surrogate?"),
    break, read(R),
    insert(rdas(R,D,A)).
 ← var(D) ∧ ¬(nonvar(A) ∧ rdas(_,_,A)),
    write("domain surrogate?"),
    break, read(D),
    insert(rdas(R,D,A)).
 ← nonvar(A) ∧ nonvar(R) ∧ nonvar(D) ∧ ¬(rdas(_,_,A)) ∧
    ¬(rdas(R,_,B) ∧ attn(A,N) ∧ attn(B,N) ∧ ¬(A=B)),
    assert(rdas(R,D,A)),
    insert(relation(R)),
    insert(domain(D)),
    insert(attribute(A)).
```

Figure 10. Insertion into rdas.

```
delete(rdas(R,D,A))
 ← ¬(rdas(R,D,A)).
 ← var(A) ∧ var(D) ∧ var(R),
    write("nothing done").
 ← nonvar(A) ∧ rdas(R,D,A) ∧ rdas(R,_,B) ∧ ¬(A=B),
    retract(rdas(R,D,A)),
    delete(attribute(A)).
 ← nonvar(A) ∧ rdas(R,D,A) ∧ ¬(rdas(R,_,B) ∧ ¬(A=B)),
    retract(rdas(R,D,A)),
    delete(attribute(A)),
    delete(relation(R)).
 ← nonvar(D) ∧ var(A) ∧ var(R) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(_,D,_)).
 ← nonvar(R) ∧ var(A) ∧ var(D) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(R,_,_)).
 ← nonvar(R) ∧ nonvar(D) ∧ var(A) ∧ rdas(R,D,A),
    delete(rdas(R,D,A)),
    delete(rdas(R,D,_)).
```

Figure 11. Deletion from rdas.

rdas	(R,	D,	A)
	-	-	a
	r	-	a
	-	d	a
	r	d	a
	-	d	-
	r	-	-
	r	d	-

Figure 12. Possible instantiated/uninstantiated parameter combinations.

```
delete(attribute(A))
 ← ¬(attribute(A)).
 ← nonvar(A) ∧ attribute(A),
    retract(attribute(A)),
    delete(rdas(_,_,A)),
    delete(attn(A,_)).
 ← var(A),
    write("attribute surrogate?"),
    break,
    read(A),
    delete(attribute(A)).
```

Figure 13. Deletion from attribute.

```
delete(relation(R))
 ← ¬(relation(R)).
 ← var(R),
    write("relation surrogate?"),
    break,
    read(R),
    delete(relation(R)).
 ← nonvar(R) ∧ relation(R),
    retract(relation(R)),
    delete(reln(_,R)),
    delete(rdas(R,_,_)).
```

Figure 14. Deletion from relation.

Farther down the intension-extension dimension

We need to control schema definition from both the metaschema level and the data dictionary schema level. The first question is therefore what the initial content of the intension-extension dimension is. The second question is how we set up the initial content of the intension-extension dimension.

As we shall see, the initial content of the data dictionary schema includes only the bare minimum needed to control the definition and change of application schemata. A data dictionary schema must do much more than that. The third question therefore is how the database administrator chooses, defines, and enforces a database management strategy.

The relations and operations in the metaschema define and control operations only on the immediate extension of the metaschema. They define and control only intralevel propagation of a schema change. Insofar as general rules and laws can be identified, which is difficult, the metaschema and the data dictionary schema should also define and control interlevel propagation caused by changing an extension interpreted as intension for the next level. The fourth question therefore is how we specify and enforce interlevel propagation.

These four questions will be addressed in the following short sections.

Initial content. We need to control the definition and change of relations and operations from both the metaschema and the data dictionary schema. In the first case, we must control the definition and change of the data dictionary schema. In the second case, we must control the definition and change of the application schemata.

The relations and operations defined in the previous section are not level specific and we can use them at any level where we need to control the definition and change of schemata at the next level. We therefore choose to include the relations and operations defined previously in both the metaschema and the data dictionary schema, also in accordance with our principle of self-description.

On the other hand, we do need to be level specific when we call and execute an operation on some relation. We choose to identify the level of data description at which a level-specific operation is defined by prefixing operation names with an m for metaschema, a dd for data dictionary schema, and an s for application schema.

A set of level-specific metaschema operations can be defined as shown in

```
delete(reln(N,A))
 ← ¬(reln(N,R)).
 ← var(N) ∧ var(R),
    write("relation name?"),
    break,
    read(N),
    delete(reln(N,_)).
 ← ¬(var(N) ∧ var(R)) ∧ reln(N,R),
    retract(reln(N,R)),
    delete(relation(R)).
```

Figure 15. Deletion from reln.

```
m_delete(reln(N,R))
  ← c1,
    delete(reln(N,R)).
```

Figure 16. Metaschema level-specific deletion from reln.

```
m_insert(reln(N,R))
  ← c2,
    insert(reln(N,R)).
    ..
    ..
```

Figure 17. Metaschema level-specific insertion into reln.

Figures 16 and 17. The purpose of the conditions c1, c2, ... is to protect from changes that part of the data dictionary schema that has the twofold role of being a stored description of the metaschema and an integral part of the data dictionary schema controlling application schema definition and change.

As a simple example of how the m-operations protect the stored description of the metaschema from modifications, consider the condition c1 in the operation specification in Figure 16. It must include the following:

$$c1 = \neg(N = name1) \wedge \neg (N = name2) \wedge \ldots \wedge \neg(N = namen),$$

where name1, ... , namen are the names of the metaschema relations.

We choose to let the initial content of the data dictionary data be an empty set of relations ready to hold the extension of the initial content of the data dictionary schema.

Finally, we choose not to include any initial content of application data.

Our choices for the initial content of the intension-extension dimension can be summarized as in Figure 18.

The metaschema consists of the metaschema relations and the operations defined previously, and level-specific metaschema operations as those defined in Figures 16 and 17. The metaschema itself is imaginary, meaning that nothing is actually stored in the dotted line box. We will later explain how the relation definitions of the metaschema itself are hard-wired in the DL-processor.

A description of the metaschema is explicitly stored in the black box of the data dictionary schema. Nothing in the black box can be changed by operations defined in the metaschema—the metaschema is self-describing, not self-destructing. The conditions in the level-specific metaschema operations prevent changes of the stored description of the metaschema. Nothing but the description of the metaschema is part of the initial content of the data dictionary schema.

The initial content of the data dictionary data is an empty set of relation tables ready to store the extension of the initial data dictionary schema.

Setting up the initial intension-extension dimension. In order to explain how we set up the initial content of the intension-extension dimension, we must first make some assumptions about the DL-processor.

The DL processor. The DL processor—the DMCS—has the definition of the metaschema built in. This basically means that it knows the names of the metaschema relations, how they are structured, and in which files their extensions are stored. The DL processor does not have the definition of the metaschema operations built in. Instead, it has a search module that retrieves an operation specification given an operation call. It locates m-operations from the data dictionary schema, dd-operations from the data dictionary schema, and s-operations from the data dictionary data.

The execution module executes non-level-specific operations relative to the level specific operations that called them. If the level cannot be decided from the call, the operation fails.

The DL processor enforces the operational semantics of the update dependency formalism. As part of this, the specification of primitive operations is built in.

The setup. To set up the initial content of the intension-extension dimension, we replace the search module of the DL processor by a booster module. The booster module searches an externally stored set of level-specific metaschema operation specifications to retrieve an operation specification, given an operation call. This externally stored set of level-specific metaschema operations differs from the set we want to store in one sense: the operations do not have the conditions c1, c2, ... , cn defined above.

We now issue the series of m-operation calls, resulting in the insertion of the metaschema description as part of the data dictionary schema.

Finally, we replace the booster module by the search module, and we are in business.

Note that an insertion into the relation relation at any level creates an empty table at the next level to hold the extension of the inserted relation definition. This is a general rule for interlevel propagation.

Expanding the data dictionary schema. The data dictionary schema must define and control all operations on data used for database management. Most importantly, the data dictionary schema must control the definition and change of application schemata, as discussed. But, in addition to that, the data dictionary schema must define and control the notions of authorization, user, schema, program, file, distribution, etc.

As the database administrator decides on a database management strategy, he or she will define relations and operations in the data dictionary schema to enforce this strategy, and the part of the data dictionary schema copied from the metaschema will gradually be expanded into a full data dictionary. This means that the level-specific operations defined in the data dictionary schema will be more complicated than those for the metaschema. As the data dictionary schema expands, the level-specific operations in the data dictionary schema must control the propagation of changes of application schemata to changes of data dictionary data controlled by the expanded data dictionary schema.

In an expanded data dictionary schema, the level-specific operations are constructed from the original metaschema operations, as illustrated in Figure 19.

The conditions c1, c2, ... , cn, can test data controlled by the expansion of the data dictionary schema and thereby distinguish alternatives for propagating changes of data, controlled by the initial data dictionary schema, into changes of data, controlled by the full data dictionary schema. The alternative sequences of implied operations on the data, controlled by the expanded data dictionary schema, are inserted in the operation specification in Figure 19, as indicated by the dots.

Note that the database administrator must use the original non-level-specific insert, delete, and modify operations when defining the level-specific data dictionary schema operations that constitute the user interface for the database designer.

It is important to realize that the only part of the data dictionary schema that cannot be changed in any way through the metaschema is the initial part controlling relation definition and change. This means that the database administrator can design the database management application to suit the particular needs of the enterprise, the same way a database designer designs an ordinary application. If the database administrator prefers a particular kind of authorization procedure, then he or she can include it. Likewise, if the database needs to be distributed, the distribution model can be defined appropriately.

In summary, the database administrator decides on the database management strategy rather than having to suffer with

an inadequate strategy forced by the database system vendor. Or, since the metaschema is explicitly stored, several independent software houses may offer off-the-shelf, plug-compatible database management strategies (that is, data dictionary schema definitions) that the database administrator can choose from. Such plug-compatible data dictionary schema definitions could only be produced by the original vendor if the metaschema was not explicitly described.

Our database system framework supports the notions of plug-compatible data and plug-compatible software. Therefore, we can strip the DMCS to the bones, leaving only the DMCS functions, which are absolutely essential. We know of no other database system or data dictionary system born this naked: they all come with more of the database management strategy built in, such as an authorization strategy, and with built-in data management tools that are nice and important, but not essential.

In summary, the simplicity and potential of our framework is based on

- the explicitly stored metaschema description that gives the plug-compatibility;
- the explicitly stored and changeable data dictionary schema that allows us to design our own database management strategy; and
- the power of the update dependency formalism that allows us to fully follow the 100 percent principle, which means that an intension completely controls its extension and thereby relieves the DL processor from enforcing a lot of special rules.

The following simple example illustrates how the database administrator may expand the data dictionary schema.

Example. The database administrator can define a simple authorization strategy for all application database users by defining the relation and operations in the data dictionary schema using the m-operations, as shown in Figure 20.

When the database designer defines an application schema relation and some operations on it, they will look like Figure 21. (We assume that the DL processor knows the username U.)

When the database designer has defined the operations using the dd-operations, he inserts tuples in the relation authorized, allowing the chosen users to do the chosen operations.

When user u1 calls an operation, such as s_insert(supplier(..)), it will succeed only if the tuple (u1,supplier,s_insert) is stored in the relation authorized.

This was the simple case, where the database administrator trusts that the database designer will remember to insert the condition on the relation authorized in all update dependencies of all s-operations defined. If the database administrator

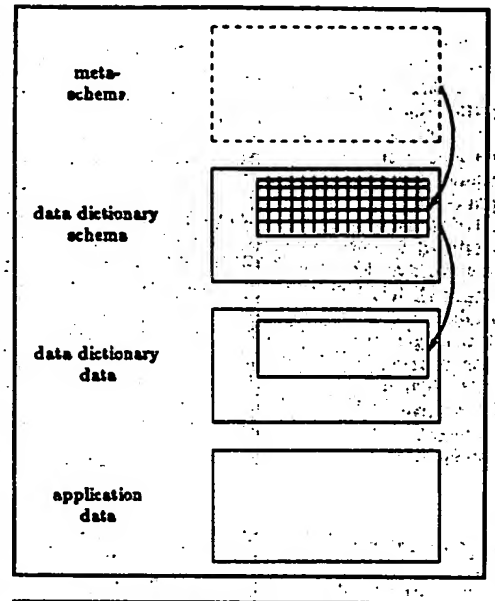

Figure 18. Initial content of the intension-extension dimension.

```
dd_delete(reln(N,R))
→ c1,
  delete(reln(N,R)),
  ..
  ..
→ c2,
  delete(reln(N,R)),
  ..
  ..
→ cn,
  delete(reln(N,R)),
  ..
  ..
```

Figure 19. Data dictionary schema level-specific deletion from reln.

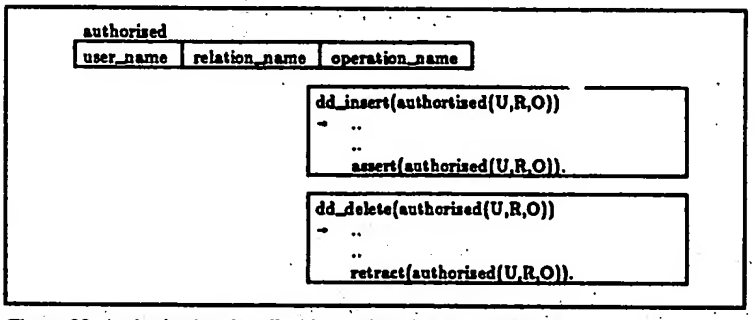

Figure 20. Authorization described in the data dictionary schema.

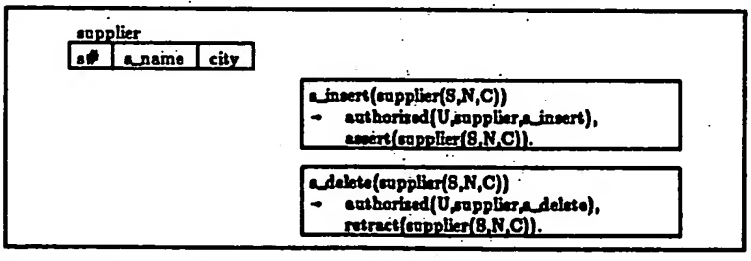

Figure 21. An example of an application schema.

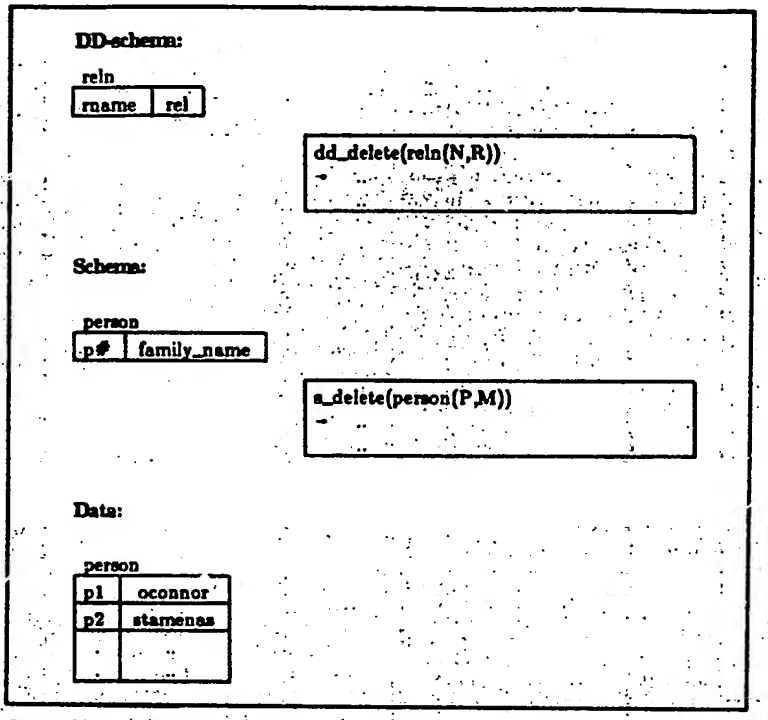

Figure 22. Interlevel propagation.

```
delete(reln(N,R))
→ reln(N,R) ∧ op_spec(O,R) ∧ opn(O,M) ∧ ¬(¬(M=dd_delete) ∧ ¬(M=s_delete)),
   write("Delete all tuples in"),
   write(N),
   write("using operation"),
   write(M),
   break,
   retract(reln(N,R)),
   delete(relation(R)).
→ ..
   ..
   ..
```

Figure 23. Instructing the user about interlevel propagation.

wants to make sure of this, he or she must define the dd-operations on the relation named condition to force all conditions of s-operations to include authorized (U, < relation_name >, < operation_name >). When the database designer later calls the dd-operations to define s-operations, there is no way to exclude or forget the condition on the relation authorized in any of the update dependencies constituting the s-operations.

Interlevel propagation. We have already given a specification of operations defining and controlling schema updates and their intralevel propagation. We now study the interlevel propagation of a schema update, that is, the effect a schema update has on the data defined and controlled by the schema.

Example. Consider Figure 22. If we want to delete the tuple (person,r1) from reln, we make the operation call dd_delete(reln(person,r1)). This operation only affects tuples in the database defining the schema. We would somehow like all the tuples in the relation person to be deleted, too.

There are three ways in which we can try to handle the problem of interlevel propagation. They are all needed. We can

- include a set of rules for interlevel propagation in the semantic definition of the DL;
- specify interlevel propagation directly in the operation specifications; or
- provide a data management tool for database reorganization.

Including interlevel propagation in the semantics of the DL. Only general rules for interlevel propagation for our data model should be included in the semantic definition of the DL.

Before we define these rules, let us illustrate one of the pitfalls of the problem of interlevel propagation by considering the deletion of an attribute from a relation with a nonempty extension. If we think that the general rule for interlevel propagation in the relational model in this situation is enforced by projecting the extension of the relation over the remaining attributes, then we are in for a surprise. The problem is not that we don't know which duplicate tuples, if any, to get rid of. The problem is that the relational projection operator has nothing to do with the process of deleting an attribute from a relation definition. In some situations we may decide that when we delete an attribute from a relation definition, the extension of the new relation should be computed by projection, but this is not a general rule for the relational model.

The reason for our surprise is that we have gotten used to the production of an artificial intension of a relation every time we use the relational operators on some relation extensions. What we must realize is that the relational operators have no intensional semantics. That is, the intension produced by the relational operaters has no meaning; it must be assigned by humans.[12]

We know of only a few general rules for interlevel propagation in the relational data model:

(1) A relation definition can be deleted if the extension of the relation is currently empty. This rule is implemented in Chamberlin[13] and Zloff.[14]

(2) A relation definition can be inserted. The extension of the relation will defined to be empty. This rule is implemented in Chamberlin[13] and Zloff.[14]

(3) An attribute definition can be deleted from a relation definition if the extension of the relation is currently empty. This rule is implemented in Chamberlin[13] and Zloff.[14]

(4) An attribute definition can be inserted in a relation definition if the extension of the relation is currently empty. This rule is implemented in Zloff.[14] It is generalized in Chamberlin,[13] where an attri-

bute definition can be inserted in a relation with a nonempty extension. The corresponding values in the extension of the relation are defined to be null, representing value unknown or value inapplicable.

(5) A domain definition can be deleted if it is not part of any relation definition. A domain definition can be deleted from a set of relation definitions if all the attributes defined over the domain can be deleted from the relation definitions. The interlevel propagation is through the deletion of the attribute definitions.

(6) A domain definition can be inserted in a relation definition if the implied attribute definition can be inserted in the relation definition. The interlevel propagation is through the insertion of the attribute definition.

(7) A view definition can be deleted without any interlevel propagation. This rule is implemented in Chamberlin.[13]

(8) A view definition can be inserted without any interlevel propagation. This rule is implemented in Chamberlin.[13]

We include the general rules (1) through (8) in the semantics of the DL. This means that when we insert a base relation definition, the system will create an empty table to hold the extension of the relation, and when we delete a base relation definition, the system will remove the empty table.

All of the above rules for interlevel propagation, except for the generalization of rule (4), actually cover situations where there are no interlevel propagation, except for the empty tables that are set up or removed.

To help the database administrator and the database designer bring the database into a state that allows a schema update, we can let the non-level-specific operations in the metaschema give us a couple of hints, as illustrated in the following example.

Example. In order to enforce a generalization of rule (1), to allow the deletion of a relation with a nonempty extension, we could change the definition of the delete operation on relation reln as illustrated in Figure 23.

The condition in the operation checks that there exists a delete operation for the level in question. The relations op_spec and opn are two of the relations in the expanded metaschema controlling the definition and change of operation specifications. The operation simply tells the user which interlevel propagation must be taken care of before passing control back to the operation.

General rules for interlevel propagation are more interesting in a data model that supports the notion of subtype or is-a relationships. We have defined a set of rules elsewhere.[15]

```
dd_delete(reln(N,R))
←
←  reln(N,R) ∧ op_spec(O,R) ∧ opn(O,s_delete) ∧ (N=person_name),
   s_delete(person_name(_,_)),
   retract(reln(person_name,R)),
   dd_delete(reln(person_address,P)),
   dd_delete(relation(R)),
←
```

Figure 24. Specifying interlevel propagation in an operation.

Specifying interlevel propagation in the operations. Data-dependent rules for interlevel propagation can be explicitly modeled in the operations. If the rules are general for the data model, then they should be included in the non-level-specific operations in the metaschema. If the rules apply to a specific application of the data model, then they should be included in the level-specific operations.

Example. Suppose we define two relations, person_name(p#, name) and person_address(p#, address), with compound update operations enforcing a referential integrity constraint from person_address to person_name. This means that when we insert the tuple (pi, address) in person_address, then a tuple (pi, name) must be present in or inserted into person_name, and vice versa for deletion.

Suppose we want to generalize this rule to the relation definitions themselves, meaning that if we delete the relation person_name, then we want to delete the definition of the relation person_address, too.

We can specify this rule in the data dictionary schema as shown in Figure 24.

The condition of the operation checks that there exists an operation with the name s_delete for the relation to be deleted. The relations op_spec and opn are two of the relations in the expanded metaschema used to model and control the definition of operations. The operation simply enforces the data-dependent rule that if the relation definition to be deleted is for the relation person_name, then all tuples in the extension of this relation must first be deleted.

This technique works very well on data-dependent rules for interlevel propagation. The technique can be used both in level-specific operations in the metaschema and the data dictionary schema.

Database reorganization. When interlevel propagation cannot be included in the semantics of the DL or explicitly specified in the operations, then we must resort to tools for database reorganization. This means that the database administrator or the database designer will be responsible for the interlevel consistency of intensions and extensions.

Database reorganization is often a very elaborate process involving a system shutdown. Only a few systems, including SystemR[13] and QBE,[14] support on-line database reorganization. A powerful algebra for database reorganization was proposed for the extended relational model RM/T.[16]

On-line database reorganization is supported by a self-describing database system. The general technique follows:

(1) Insert the definition of the new set of relations.

(2) Insert the definition of the compound update operations for the new relations.

(3) Write a data management tool that uses the delete operations on the old relations and the insert operations on the new relations to move and call the data.

(4) Delete the definitions of the old relations.

(5) Rename the new relations.

This completes our discussion of the intension-extension dimension. We have described its initial contents and how to set it up. We have explained how to expand the data dictionary schema. And, we have discussed three ways of handling interlevel propagation.

In this article we concentrated on the metadata management aspects of a self-describing database system, and we used the relational data model and the formalism for update dependencies to specify the conceptual level of a self-describing database system.

The architecture for self-describing database systems has been accepted by

ANSI/SPARC and is being considered by the ISO as a reference model for database systems in the late 1980's and 1990's. We will undoubtedly see several database systems develop in this direction.

We are ourselves currently using a self-describing database system in the design of a system for scientific information interchange.

References

1. L. Mark and N. Roussopoulos, "The New Database Architecture Framework—A Progress Report," *Proc. IFIP WG 8.1 Working Conf. Theoretical and Formal Aspects of Info. Systems*, North-Holland, 1986.
2. T. Burns et al., "Reference Model for DBMS Standardization," Report from the Database Architecture Framework Task Group of the ANSI/X3/SPARC Database System Study Group, Sigmod Records, Mar. 1986.
3. D. Tsichritzis and A. Klug (eds.), "The ANSI/X3/SPARC DBMS Framework," *Info. Systems*, Vol. 3, No. 3, 1978.
4. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, Vol. 13, No. 6, 1970.
5. B. Breutman, E. Falkenberg, and R. Mauer, "CSL, A Language for Defining Conceptual Schemas," *Database Architecture*, G. Bracchi and G. M. Nijssen (eds.), North Holland, 1979.
6. G. M. Nijssen, "One, Two or Three Conceptual Schemata," *Database Architecture*, G. Bracchi and G. M. Nijssen (eds.), North Holland, 1979.
7. P. Hall, J. Owlett, and S. Todd, "Relations and Entities," *Modelling in Data Base Management Systems*, G. M. Nijssen (ed.), North Holland, 1976.
8. J. J. van Griethuysen (ed.), "Concepts and Terminology for the Conceptual Schema and Information Base," ISO/TC97/SC5/WG3—N695, Philips, 1982.
9. L. Mark, "Self-Describing Database Systems—Formalization and Realization," PhD dissertation, TR-1484, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, 1985.
10. L. Mark and N. Roussopoulos, "Operational Specification of Update Dependencies," submitted to the *ACM Trans. Database Systems*, 1986.
11. R. Reiter, "On Closed-World Data Bases," Logic and Data Bases, H. Gallaire and J. Minker (eds.), Plenum Press, 1978.
12. N. Roussopoulos, "Intensional Semantics of a Self-Documenting Relational Model," TR-1264, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, 1982.
13. D. D. Chambertin et al., "Sequel 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. Research and Development*, Vol. 20, No. 6, 1976.
14. M. M. Zloff, "Query-by-Example: A Data Base Language," *IBM Systems J.*, Vol. 16, No. 4, 1977.
15. N. Roussopoulos and L. Mark, "Schema Manipulation in Self-Describing and Self-Documenting Data Models," *Int'l J. Computer and Info. Sciences*, Vol. 14, No. 1, 1985.
16. E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Trans. Database Systems*, Vol. 4, No. 4, 1979.

Nick Roussopoulos' biography and photograph appear following the Guest Editor's Introduction.

Leo Mark is an assistant professor in the department of computer science at the University of Maryland. His research interests include database management systems architecture, data models, database semantics, database integrity, metadata management, and data dictionary systems.

Mark received his Masters in computer science from Aarhus University, Denmark, in 1980. He first joined the Database Systems Group in the department of computer science at the University of Maryland as a student in January of 1983. There he participated in database architecture standardization efforts for ANSI/SPARC and a research project on a standardized communication environment for NASA. He received his PhD from Aarhus University in 1985.

Readers may write to the authors at the Department of Computer Science, University of Maryland, College Park, MD 20742.